

Lecture 25: Transactions

mass-production of database updates

- what is a transaction?
- potential problems
- recoverability
- transaction processing schedules
- transaction processing schedule properties
- locking for schedule serialisability
- logging strategies and recovery

[E&N, mostly Chapter 17, but 18 and 19 too]

Announcements

- (Not sure I should be advertising this, but...)
- Tomorrow (Friday 26th September) is the last day you can drop a course and receive **WD** (withdrawn without failure) instead of **WN** (withdrawn with failure), and avoid tuition fees.
- However, ISIS is down from 5pm today, so do it online by then or fill out a paper course variation form tomorrow.
- Assignment 2 will appear on the course website on the first day back from break: Monday 13 October. It will be due at 5pm Friday 24 October.
- Mid-semester exam results will be available on streams tomorrow
- Ben will be taking the remaining lectures, I'm off to Philosophy at Macquarie Uni Sydney!

Assignment 1

Marked assignments for the following groups are at the front for you to collect:

- Monday 3pm
- Monday 5pm
- Tuesday 9am
- Tuesday 11am
- Tuesday 3pm
- Wednesday 5pm

Some are not yet marked

- Friday 9am: these will be left at the DCS office by Monday for collection
- Monday 1pm and Thursday 9am, not yet marked
- Students who were granted extensions, not yet marked

Minimal Cover - The Hard Part!

[E&N, §10.2.4]

A minimal cover of a set of functional dependencies E is an *equivalent* set of functional dependencies F such that

- 1 each dependency is in the form $X \longrightarrow A$ (right hand side is a single attribute)
- 2 we cannot replace any dependency $X \longrightarrow A$ with $Y \longrightarrow A$ where $Y \subset X$ and remain equivalent to E eg. replace $BCD \longrightarrow A$ with $CD \longrightarrow A$
- 3 we cannot remove any dependency and remain equivalent to E

How do we ensure the second condition?

Minimal Cover - The Hard Part!

$$BC \longrightarrow D \models ABC \longrightarrow D$$

(removing LHS attribute makes dependency *stronger*)

$$\text{therefore } \{CD \longrightarrow A, B \longrightarrow C\} \models \{BCD \longrightarrow A, B \longrightarrow C\}$$

the two sets are equivalent iff

$$\{BCD \longrightarrow A, B \longrightarrow C\} \models \{CD \longrightarrow A, B \longrightarrow C\}$$

$$\text{and that is so iff } \{BCD \longrightarrow A, B \longrightarrow C\} \models CD \longrightarrow A$$

Determine this by trying to derive it using the rules[§10.2.2], or drawing a diagram and following the arrows.

Transactions

So far, we have talked a lot about queries, but databases must also be updated.

Many big databases need to be updated a *lot*: many times a second, by many different users.

- airline and hotel chain reservations
- banks
- chain store checkouts
- securities trading

For each of these kinds of application, its the same few kinds of updates being repeated a lot.

DBMS systems support the idea of a *transaction* to enable this kind of application.

Minimal Cover - The Hard Part!

So, what you need to do is this

for each FD $X \longrightarrow A \in E$ with $|X| > 1$

for each $B \in X$

$$Y := X - \{B\}$$

if you can derive $E \models Y \longrightarrow A$

then replace $X \longrightarrow A$ with $Y \longrightarrow A$ (new E)

If you can *prove* that you can't derive something, that's excellent, but don't worry too much about that.

A Simplified Data-Model (and some notation, yay!)

The theory of transactions is easier to follow if we simplify our data model.

Instead of the full relational model, we just consider

- named data items X, Y, Z, \dots
- same names used for disk storage and program variables
- each transaction execution has its own copies of the variables
- operations: read r and write w
- so $r_4(X)$ sets transaction execution 4's variable X to the value of X on disk

Transaction Programs, Executions and Schedules

a transaction program to pay interest might be written as

$$r(X); X := X * 1.1; w(X)$$

The DBMS will have many transactions to process, so it needs to consider many operations from many different transactions, and how to *schedule* them.

For example, one way to pay two people's interest might be

$$r_1(X); w_1(X); r_2(Y); w_2(Y)$$

(scheduler ignores calculations, it just worries about the IO)

Often when we talk about a “transaction”, we mean a *transaction execution*, not a transaction program.

Potential Scheduling Problems - Lost Update

- if two transactions work simultaneously to update the same bank balance ...
- transaction 1 deposits \$100
- transaction 2 withdraws \$100

operation	disk value afterwards
$r_1(X);$	$X = 500$
$r_2(X);$	$X = 500$
$w_2(X);$	$X = 500 - 100 = 400$
$w_1(X);$	$X = 500 + 100 = 600$

- the effect of the withdrawal is “lost”
- the bank does not like this transaction schedule!

Database Undo!

A transaction must be all done, or not done at all.
ie transaction processing should be *atomic*

Things can go wrong after processing of a transaction has begun

- system crash
- media failure
- program logic error
- business logic error
- dbms chooses to abandon this execution and try again later (so it can do something else now)

therefore, it must have a way to “undo” partially completed transactions. This is often called a transaction “roll-back”.

Potential Scheduling Problems - Dirty Read

- banks sometimes charge less fees on accounts with a high balance
- if a withdrawal transaction begins, updates the balance reducing it below the fee-free threshold
- at that point in time, the fee calculation transaction reads the balance
- then the withdrawal transaction is rolled-back
- then the fees will be charged, incorrectly

Potential Scheduling Problems - Incorrect Summary

- suppose a company's annual salary review is seen as a single transaction
- while it's partially complete, a query calculates the total annual salary
- the result will be wrong: this total was *never* the annual salary total
- the “atomicity” of the transaction means that the business “jumped” from the “before” salaries to the “after” salaries, any intermediate situations are invalid

Conflicting Operations

There is a common pattern in each of the problem situations we just reviewed.

Two database operations are *conflicting* when

- they belong to two different transactions
- they access the same data-item
- one or both of them writes the data-item

In our examples:

- | | |
|---------------------------|---|
| lost update: | deposit and withdrawal use bank balance X |
| dirty read: | withdrawal and fee calculation use bank balance |
| incorrect summary: | some salaries used by both review and summary |

An Aside: Representation Again

- note the conflict here between transaction as
 - a unit of database processing
 - a unit of business activity
- the database technicians may say, hey, don't make such a huge single operation into a transaction, it will time-out the checkpoints!
- but the business does not want to give some people their raise and not others, nor even give some people their raise before some others - it sees it as a single all-or-nothing action
- just like database states and the real world, we have a technical idea used to represent a real-world idea
- sometimes we have to squeeze things to make them fit

Transaction Packaging and Management

A transaction consists of reads and writes, but for it to be correctly managed by the dbms, it needs some more information.

- the reads and writes need to be identified as a single transaction
- `SET TRANSACTION` is the SQL that marks the beginning of a transaction program
- the transaction can be ended by a `COMMIT`
- or `ROLLBACK` if it wants to give up and undo any changes it has made
- when studying transaction schedules, we will consider rollback and commit to be additional transaction operations
- write a_1 for rollback (abort) of transaction 1
- and c_1 for commit of transaction 1

See [E&N §17.6] for SQL code to define a transaction program

DBMS Log

The dbms maintains a *log* or *journal* of the transaction operations it performs, so that it can undo them when it needs to. (“rollback”)

The log can also be used to *redo* transactions, if eg. the database must be restored from backups. (“roll-forward”)

The log will contain the following entry types

- start transaction *transactionID*
- write *transactionID, dataItem, oldValue, newValue*
- read *transactionID, dataItem*
- commit *transactionID*
- abort *transactionID*

Question: How could you safely use the *oldValue* to undo a transaction?

Schedules

[E&N §17.4]

A **schedule** (or **history**) S of n transactions

T_1, T_2, \dots, T_n is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S , the operations of T_i in S must appear in the same order in which they occur in T_i .

We just saw a schedule $S = r_1(X); r_2(X); w_2(X); w_1(X)$; of $T_1 = r(X); w(X)$ and $T_2 = r(X); w(X)$;

A schedule can also contain a *commit* c_i or *abort* a_i operation for each transaction i .

Schedules and Recoverability

The problem is this

- the dbms has a lot of transaction executions to process
- it must decide in what order to process the operations of these transactions
- if it gets this decision wrong, bad things can happen
 - incorrect results (eg lost updates)
 - transactions can not be undone
 - undoing a transaction might require undoing a lot of other work too (next time)

We want some theory that tells us which transaction processing schedules are good, and which ones we should avoid.

Commit and Schedule Recoverability

Once the commit operation has been executed for a transaction, it is not to be rolled-back.

But consider this schedule

$$r_1(X); w_1(X); r_2(X); w_2(X); c_2; a_1$$

Transaction 2 used the updated information left there by transaction 1, but now we are backing out of that one.

Therefore we must back out of transaction 2 as well, but it is committed.

We call a schedule like this *unrecoverable*.

Recoverable

A transaction T **reads** from transaction T' in a schedule S if some item X is first written by T' and later read by T .

[E&N §17.4.2]

A schedule is recoverable if the commit or abort operation of every transaction is *after* the commit or abort operation of every transaction it reads from.

Strict

(Remember the question about using the the “before” value in write records in the dbms log?)

$w_1(X, 5); w_2(X, 8); a_1$

Suppose that the value of X was originally 9, which is the before image stored in the system log along with the new $w_1(X, 5)$ operation. ... the recovery procedure ... will restore the value of X to 9, even though it has already been changed to 8 by transaction T_2 ...

A *strict* schedule allows us to use this dumb recovery procedure safely. It prohibits overwriting values written by uncommitted transactions.

Cascadeless

This schedule is recoverable

$r_1(X); w_1(X); r_2(X); w_2(X); a_1$

But backing out of T_1 requires also backing out of T_2 .

This might extend back through lots of transactions, leaving a lot of work to redo.

What allowed this problem in the example?
Reading from an uncommitted transaction!

A **cascadeless** schedule is one where if T reads from T' , then T' is committed.

Serialisability

Recoverable, cascadeless and strict are increasingly strong properties relating to recovery.

But we have not yet characterised which schedules are “correct”.

This is the point of the “serialisable” schedule property. [E&N §17.5]

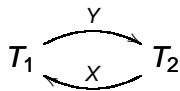
Serial

- transactions are intended to be completed as a unit
- if we run all of one transaction, then all of another etc, everything will be fine: that is called a *serial* schedule
- except we will waste LOTS of time waiting for disk IO
- schedules which are somehow “equivalent” to a serial schedule are correct
- but what does “equivalent” mean?

See [E&N Figure 17.5(a) and (c)] (whiteboard).

(Conflict) Serialisable

- A schedule is *conflict serialisable* if it is conflict equivalent to a serial schedule
- or we can just call it *serialisable*
- this can be tested using a fun picture-drawing technique. . .



Conflict Equivalent

- “result equivalence” is discussed and rejected in [E&N §17.5] (are you convinced by that discussion?)
- two operations are conflicting if they
 - come from different transactions
 - access the same data-item
 - at least one of them is a write
- two schedules for the same set of operations are *conflict equivalent* if each pair of conflicting operations appears in the same order in both schedules
- Can the results of two conflict equivalent schedules be different?

Precedence Graph to Test Serialisability

[E&N Figure §17.7, and §17.5 again]
also [E&N Algorithm 17.1]

- a *directed graph* is a bunch of points with arrows between them
 - directed graphs are also called *digraphs* or even just *graphs*, the points are called *nodes* or *vertices*, the arrows *edges* or *arcs*
- we can draw a graph showing the ordering of conflicting operations
- each transaction is a node
- each ordered pair of conflicting operations is an arrow
- the schedule is serialisable iff there are no cycles

Locking

[E&N Chapter 18, (just the beginning bit)]

How do we ensure that dbms schedules are serialisable?

One way is to maintain a system of *locks* on the data-items.

- a *lock* is variable which controls the accessibility of a data-item
- transactions must perform *lockItem(X)*, *unlockItem(X)* operations
 - and perhaps some others
 - but actually, the dbms will usually do this implicitly
 - the lock operations must be implemented so they are not preempted by other processes, ie they are OS *critical sections*
- an efficient implementation: only store variables for locked items, using hashed storage

2-Phase Locking

The problem there is that T_1 left things unlocked while it was half-way through its work.

The two phase locking protocol (2PL):

- first phase - locking
- second phase - unlocking
- ie the transactions set of locked items strictly increases, then strictly decreases
- (similarly when we have read/write locks)
- if all transactions follow 2PL, then all schedules are serialisable
- however, we can get *deadlocks* - where several transactions are each waiting for something locked by another
- many variations of 2PL, some avoid deadlock

Binary Locking

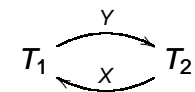
An impractical but easy to understand locking system:

- locks are binary
- *lockItem(X)* locks it if its unlocked, otherwise requesting transaction sleeps till item unlocked
- *unlockItem* operation unlocks it and wakes a waiting transaction if there are any
- transactions must play by the rules: lock before use, unlock after use, don't lock what you already locked, don't unlock what you haven't locked
- but **this locking scheme does not ensure serialisability** (consider [E&N Figure 18.3])

Serialisability Requires *Two-Phase* Locking eg.

T_1	T_2
readLock(Y)	
readItem(Y)	
unlock(Y)	
	readLock(X)
	readItem(X)
	unlock(X)
	writeLock(Y)
	readItem(Y)
	$Y := X + Y$
	writeItem(Y)
	unlock(Y)
writeLock(X)	
readItem(X)	
$X := X + Y$	
writeItem(X)	
unlock(X)	

[E&N Figure 18.3]
serialisation graph for this schedule:



its not serialisable:
 T_1 does not obey the *two phase* locking protocol
(nor does T_2)

Physical Transaction Processing

So far we have considered transaction schedules in a rather abstract way, ignoring the complications of computer hardware and operating systems.

In practice, the dbms must manage memory *buffers* where the updates are done.

The strategy for writing the updated buffers back to disk determines what recovery strategies are possible.

Checkpoints

Most of the time there will be “*dirty*” buffers in memory - buffers that have been updated, but not yet written back to disk.

The dbms will try to periodically achieve a state where all updates are written, and write a *checkpoint* record to the log. These records are important for recovery.

Caution: “checkpoint” is used in slightly different ways in different systems and texts. eg. In the Model204 dbms, checkpoints are only written when there are no active (partially complete) transactions.

What a checkpoint “means” depends on the dbms update strategy.

Recovery and the Log

There are two main recovery scenarios

catastrophic the database files are lost
(usually due to hardware failure)

non-catastrophic the dbms crashed, but the files are intact

The dbms log (or journal) is essential for most recovery, and special measures will often be taken to keep it safe.

- writing it to several devices simultaneously
- frequent backups
- off-site copies

Log entries must be written *before* the updates they record. This is called *write-ahead logging*.

Catastrophic Failure

In the “catastrophic” case, database files must be recovered from backups.

The journal is used to replay the transactions (*or roll-forward* the database).

Except we do not want the updates of partially completed transactions. Must either analyse the log first, or roll-forward everything, then roll-back the transactions that are active at the end of the log.

Immediate Update

The most common dbms update strategy is also the most complicated to recover from, but allows greater concurrency, better memory management and less disk IO.

- dirty buffers can be written any time (before or after the commit of the transactions the updates belong to)
- therefore, pages can be “stolen” if memory is needed for something else
- commit records in the log only tell us that the whole transaction has been *logged*
- to recover, must **undo** write operations of uncommitted transactions
- and **redo** write operations of committed transactions

Deferred Update

Deferred update is another strategy, where updates are not allowed to be applied till *after* the commit record is written.

- may require a lot of memory!
- never need to undo operations of uncommitted transactions
- but may have to redo operations of committed transactions
- commit record means transaction application *may* have begun!

Forced Immediate Update

a variation of the immediate update strategy avoids the need to redo stuff

- all transaction updates must be applied *before* the commit record is written
- therefore, the commit record means all updates have been applied

Transaction Summary

- what is a transaction?
- potential problems
- recoverability
- transaction processing schedules
- transaction processing schedule properties
- locking for schedule serialisability
- logging strategies and recovery

Have a nice break!